http://www.labo-sun.com
labo-sun@supinfo.com

LABORATOIRE SUPINFO DES TECHNOLOGIES Sun

RMI

APPLICATION DISTRIBUEE

Auteur : Renaud Roustang
Version 2.0 – 31 janvier 2005
Nombre de pages : 19

Paris Academy of Computer Science
SUPINFO Paris
Ecole Supérieure d'Informatique

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. Introduction

Remote Method Invocation (RMI) permet la création et l'utilisation d'applications Java distribuées de manière aisée.

1.1. Applications distribuées

Une application distribuée est une application dont les classes sont réparties sur plusieurs machines différentes. Dans de telles applications, on peut invoquer des méthodes à distance. Il est alors possible d'utiliser les méthodes d'un objet qui n'est pas situé sur la machine locale.

Déjà dans le langage C, il était possible de faire de l'invocation à distance en utilisant RPC (*Remote Procedure Calls*). RPC étant orienté "structure de données", il ne suit pas le modèle "orienté objet".

RMI (*Remote Method Invocation*) permet l'utilisation d'objets sur des JVM distantes. Il va plus loin que RPC puisqu'il permet non seulement l'envoi des données d'un objet, mais aussi de ses méthodes. Cela se fait en partie grâce à la sérialisation des objets (cf. cours Input/Output). Il est également possible de charger le byte-code des classes dynamiquement.

Il existe une technologie, non liée à Java, appelée CORBA (*Common Object Request Broker Architecture*) qui est un standard d'objet réparti. CORBA a été conçu par l'OMG (*Object Management Group*), un consortium regroupant 700 entreprises dont Sun. Le grand intérêt de CORBA est qu'il fonctionne sous plusieurs langages, mais il est plus lourd à mettre en place.

RMI est un système d'objets distribués plus simple que CORBA. Nous ne traiterons pas ce dernier dans ce cours. Sachez tout de même qu'il existe l'IIOP (*Internet InterObject Protocol*) qui permet l'interopérabilité entre RMI et CORBA.

1.2.RMI

Le but de RMI est de créer un modèle objet distribué Java qui s'intègre naturellement au langage Java et au modèle objet local. Ce système étend la sécurité et la robustesse de Java au monde applicatif distribué. RMI permet aux développeurs de créer des applications distribuées de manières simples puisque la syntaxe et la sémantique reste la même que pour les applications habituelles.

RMI est apparu dès la version 1.1 du JDK et a été étoffé pour la version 2.

Avec RMI, les méthodes de certains objets (appelés objets distants) peuvent être invoquées depuis des JVM différentes (espaces d'adressages distincts) de celles où se trouvent ces objets, y compris sur des machines différentes via le réseau. En effet, RMI assure la communication entre le serveur et le client via TCP/IP et ce de manière transparente pour le développeur.

Il utilise des mécanismes et des protocoles définis et standardisés tels que les sockets et RMP (*Remote Method Protocol*). Le développeur n'a donc pas à se soucier des problèmes de niveaux inférieurs spécifiques aux technologies réseaux.

L'architecture RMI définit la manière dont se comportent les objets, comment et quand des exceptions peuvent se produire, comment gérer la mémoire et comment les méthodes appelées passent et reçoivent les paramètres.

RMI préserve la sécurité inhérente à l'environnement Java notamment grâce à la classe ***RMISecurityManager*** et au DGC (*Distibuted Gabage Collector*).

2. Architecture RMI

2.1. Interfaces

L'interface est le cœur de RMI. L'architecture RMI est basé sur un principe important : la définition du comportement et l'exécution de ce comportement sont des concepts séparés.

RMI permet au code qui définit le comportement et au code qui implémente ce comportement de rester séparé et de s'exécuter sur des JVM différentes. La définition d'un service distant est codée en utilisant une interface Java. L'implémentation de ce service distant est codée dans une classe.

Par conséquent, la compréhension de RMI réside dans le fait que les interfaces définissent le comportement et les classes définissent l'implémentation.

RMI supporte deux types de classe qui implémentent la même interface. La première est l'implémentation du comportement et s'exécute du côté serveur. La deuxième agit comme un proxy pour le service distant et s'exécute sur le client.

Un programme client crée des appels de méthodes sur le proxy, RMI envoie la requête à la JVM distante et la transfère à l'implémentation. Toutes les valeurs de retours fournies par l'implémentation sont retournées au proxy puis au programme client.

2.2. Les différentes couches de RMI

RMI est essentiellement construit sur une abstraction en trois couches.

2.2.1. Stubs et Skeletons

La première couche est celle des *Stubs* et *Skeletons*. Cette couche intercepte les appels de méthodes lancés par le client à l'interface de référence et redirige ces appels à un service RMI distant. Cette structure évite au programmeur de devoir communiquer explicitement avec l'objet distant.

2.2.2. Remote Reference Layer

La couche suivante est la couche de référence distante. Cette couche comprend comment interpréter et gérer les références faites du client vers les objets du service distant. Elle permet l'obtention d'une référence d'objet distant à partir de la référence locale (le stub).

Dans le JDK 1.1, cette couche connecte les clients aux objets du service distant qui sont exécutés et exportés sur un serveur. Cette connexion est *unicast* (point à point).

Dans le JDK 2, cette couche a été améliorée pour soutenir l'activation des objets du service distant en sommeil via la classe **RemoteObjectActivation**. Elle assure une référence persistante vers des objets distants (reconnexion éventuelle).

Ce service est assuré par le lancement du programme ***rmiregistry***.

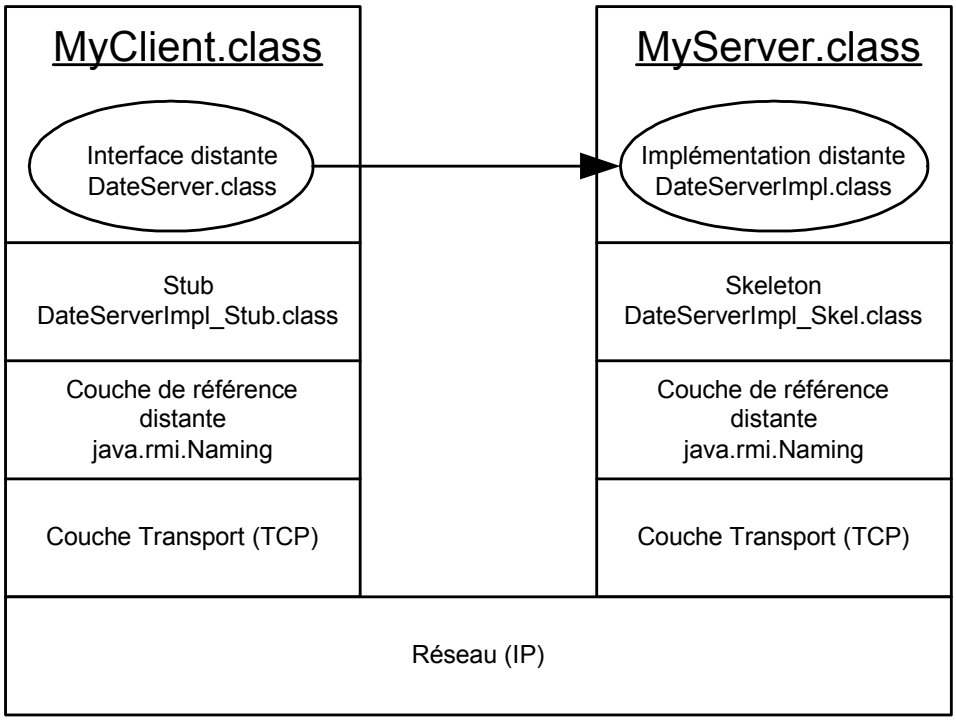
2.2.3. Couche Transport

La couche transport est basée sur les connexions TCP/IP entre les machines. Elle fournit la connectivité de base entre les 2 JVM.

De plus, cette couche fournit des stratégies pour passer les firewalls. Elle suit les connexions en cours. Elle construit une table des objets distants disponibles. Elle écoute et répond aux invocations.

Cette couche utilise les classes **Socket** et **SocketServer**. Elle utilise aussi un protocole propriétaire R.M.P. (*Remote Method Protocol*).

En utilisant une architecture en couche, chaque couche peut être augmentée ou remplacée sans affecter le reste du système. Voici un schéma montrant le transfert d'informations entre chaque couche :



2.3. Utilisation de RMI

Nous allons voir les principales étapes nécessaires à la mise en place d'un service de type RMI.

2.3.1. L'interface

La première étape consiste à créer une interface distante qui décrit les méthodes que le client pourra invoquer à distance.

Pour que ses méthodes soient accessibles par le client, cette interface doit hériter de l'interface **Remote**. Toutes les méthodes utilisables à distance doivent pouvoir lever les exceptions de type **RemoteException** qui sont spécifiques à l'appel distant.

Cette interface devra être placée sur les deux machines (serveur et client).

2.3.2. L'implémentation

Il faut maintenant implémenter cette interface distante dans une classe. Par convention, le nom de cette classe aura pour suffixe *Impl*.

Notre classe doit hériter de la classe **java.rmi.server.RemoteObject** ou de l'une de ses sous-classes. La plus facile d'utilisation étant **java.rmi.server.UncicastRemoteObject**.

C'est dans cette classe que nous allons définir le corps des méthodes distantes que pourront utiliser nos clients. Evidement, il est possible d'ajouter d'autres méthodes mais les clients ne pourront y accéder et donc ne pourront les utiliser.

2.3.3. Stubs et Skeletons

Lorsque notre client fera appel à une méthode distante, cet appel sera transféré au ***stub***. Le *stub* est donc un **relais du côté client**. Il devra donc être placé sur la machine cliente.

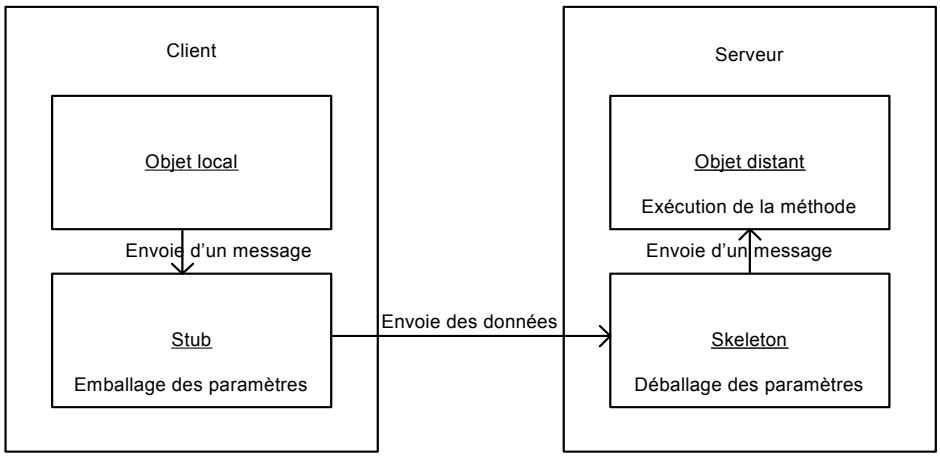
C'est le représentant local de l'objet distant. Il « marshalise » (emballe) les arguments de la méthode distante et les envoie dans un flux de données. D'autre part, il « démarshalise » (déballe) la valeur ou l'objet de retour de la méthode distante. Il communique avec l'objet distant par l'intermédiaire du *skeleton*.

Le ***skeleton*** est lui aussi un relais mais du **côté serveur**. Il devra être placé sur la machine servant de serveur. Il « démarshalise » les paramètres de la méthode distante, les transmet à l'objet local et « marshalise » les valeurs de retours à renvoyer au client.

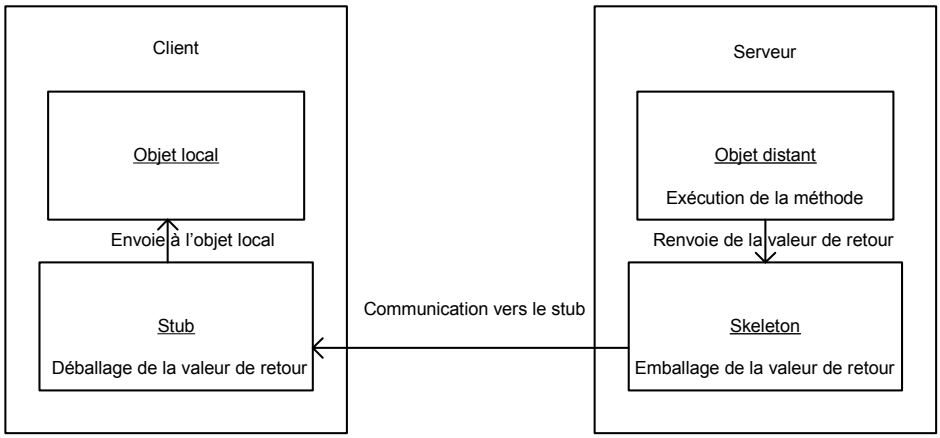
Les *stubs* et les *skeletons* sont donc des intermédiaires entre le client et le serveur qui gèrent le transfert distant des données.

On utilise le compilateur **rmic** pour la génération des *stubs* et des *skeletons*. C'est un utilitaire fourni avec le JDK

Depuis la version 2 de Java, le *skeleton* n'existe plus. Seul le *stub* est nécessaire du côté client mais aussi du côté serveur.



Invocation d'une méthode distante



Renvoie de la valeur de retour

2.3.4. Le registre RMI

Les clients trouvent les services distants en utilisant un service d'annuaire activé sur un hôte connu avec un numéro de port connu. RMI peut utiliser plusieurs services d'annuaire, y compris *Java Naming and Directory Interface* (JNDI).

Il inclut lui-même un service simple appelé Registry (rmiregistry).

Le registre est exécuté sur chaque machine qui héberge des objets distants (les serveurs) et accepte les requêtes pour ces services, par défaut sur le port 1099.

Un serveur crée un service distant en créant d'abord un objet local qui implémente ce service. Ensuite, il exporte cet objet vers RMI. Quand l'objet est exporté, RMI crée un service d'écoute qui attend qu'un client se connecte et envoie des requêtes au service. Après l'exportation, le serveur enregistre l'objet dans le registre de RMI sous un nom public qui devient accessible de l'extérieur.

Le client peut alors consulter le registre distant pour obtenir des références à des objets distants.

2.3.5. Le serveur

Notre serveur doit enregistrer auprès du registre RMI l'objet local dont les méthodes seront disponibles à distance. Cela se fait grâce à l'utilisation de la méthode statique ***bind()*** de la classe **Naming**. Cette méthode permet d'associer (enregistrer) l'objet local avec un synonyme dans le registre. L'objet devient alors disponible par le client.

```
ObjetDistantImpl od = new ObjetDistantImpl();
Naming.bind("serveur", od);
```

2.3.6. Le client

Le client peut obtenir une référence à un objet distant par l'utilisation de la méthode statique ***lookup()*** de la classe **Naming**. Il peut ensuite invoquer des méthodes distantes sur cet objet. La méthode ***lookup()*** sert au client pour interroger un registre et récupérer un objet distant. Elle prend comme paramètre une URL qui spécifie le nom d'hôte du serveur et le nom du service désiré. Elle retourne une référence à l'objet distant. La valeur retournée est du type **Remote**. Il est donc nécessaire de

caster cet objet en l'interface distante implémentée par l'objet distant. En effet, le client travaille avec l'interface distante et non avec l'objet distant lui-même.

```
ObjetDistant od = (ObjetDistant) Naming.lookup("//172.16.X.X/serveur");
```

2.3.7. Lancement

Il est maintenant possible d'exécuter l'application. Cela va nécessiter l'utilisation de trois consoles.

La première sera utilisée pour activer le registre. Pour cela, vous devez exécuter l'utilitaire **rmiregistry**.

Dans une deuxième console, exécutez le serveur. Celui-ci va charger l'implémentation en mémoire, enregistrer cette référence dans le registre et attendre une connexion cliente.

Vous pouvez enfin exécuter le client dans une troisième console.

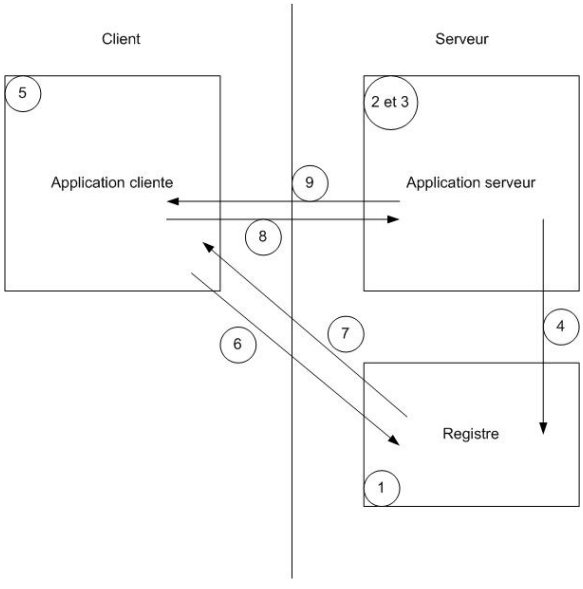
Même si vous exécutez le client et le serveur sur la même machine, RMI utilisera la pile réseau et TCP/IP pour communiquer entre les JVM.

2.3.8. Répartitions des classes

Dans un cas concret d'application client/serveur la répartition doit se faire de la manière suivante :

- Le client doit avoir accès aux interfaces distantes et aux Stubs de leurs implémentations.
- Le serveur doit avoir accès aux interfaces distantes, aux implémentations et aux Stubs.

2.3.9. Schéma récapitulatif



1. On exécute rmiregistry
2. On exécute le Serveur
3. On crée l'objet distant
4. On l'enregistre dans l'annuaire avec la méthode ***Naming.bind()***
5. On exécute le Client
6. On recherche un objet distant avec la méthode ***Naming.lookup()***
7. L'annuaire renvoie une référence de l'objet distant
8. On invoque une méthode distante
9. On reçoit une valeur de retour

3. Exemple : Serveur de date

Nous allons voir ici un exemple concret en suivant les étapes décrites dans le chapitre précédant. Le but de cette application est d'utiliser l'appel de méthodes distantes avec un simple serveur de date. Celui-ci permet aux clients de déterminer la date et l'heure du serveur.

3.1.L'interface DateServer

La seule méthode distante nécessaire à notre application est une méthode qui renvoie un objet de type **java.util.Date**. On construit donc notre interface de la manière suivante :

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface DateServer extends Remote {
	Date getDate() throws RemoteException;
}
```

Les points importants à retenir sont que notre interface étend de l'interface **Remote**. Cela permet de conférer un comportement distant aux classes qui l'implémenteront. De plus, notre méthode ***getDate()*** peut lever des exceptions de type **RemoteException**. Si nous avions déclaré d'autres méthodes, elles auraient aussi dû inclure cette clause pour être accessibles par le client. La méthode ***getDate()*** renvoie un objet de type **java.util.Date** qui implémente l'interface **java.io.Serializable**. Tous les objets de retours d'une méthode distante doivent implémenter cette interface pour transiter sur le réseau.

En effet, une méthode distante peut renvoyer trois types de valeur :

- Un type primitif (**boolean**, **int**, **float**, …)
- Un objet implémentant l'interface **Serializable**
- Un objet distant

3.2.L'implémentation DateServerImpl

La classe **DateServerImpl** implémente notre interface **DateServer** que nous avons déclarée précédemment. Elle étend la classe **UnicastRemoteObject** qui est une sous-classe de la classe **RemoteObject** et qui permet de faire la liaison avec le système RMI. Il faut fournir un constructeur en déclarant qu'il peut lever des **RemoteException**.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class DateServerImpl extends UnicastRemoteObject implements DateServer {

	public DateServerImpl() throws RemoteException{}

	public Date getDate() throws RemoteException {
		return new Date();
	}
}
```

C'est dans cette classe que nous définissons la méthode ***getDate()*** qui ne fait que retourner un objet de type **java.util.Date**. Cela est possible car l'objet **Date** est sérialisable.

3.3.MyServer

La classe Server va créer un objet de type **DateServerImpl**. Ensuite cet objet sera enregistrer auprès du Registre sous le nom **MyServer**, via la méthode ***rebind()*** de la classe **Naming**.

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class MyServer {

	public static void main(String[] args) {
		try {
			DateServerImpl ds = new DateServerImpl();
			Naming.rebind("rmi://localhost/MyServer", ds);
		} catch (RemoteException re) {
			re.printStackTrace();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Il est aussi possible d'utiliser la méthode ***bind()***, mais ***rebind()*** est plus efficace car elle peut réenregistrer l'objet auprès de l'annuaire si le nom existe déjà.

3.4.MyClient

Le client va afficher sur la console la date que lui a renvoyé le serveur. Pour cela on crée un objet de type **DateServer** (notre interface) en appelant l'objet du registre **MyServer**. Puis on appelle la méthode distante ***getDate()***.

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class MyClient {

	public static void main(String[] args) throws RemoteException {
		new MyClient(args[0]);
	}

	public MyClient(String host) {
		try {
		DateServer server = (DateServer) Naming.lookup("rmi://" + host +
"/MyServer");
		System.out.println("Date du serveur : " + server.getDate());
		}
		catch (Exception e) {
			e.printStackTrace();
		}
	}

}
```

3.5.Compilation

Pour utiliser **javac** et **rmic**, vérifier que le chemin des exécutables se trouve bien dans la variable d'environnement **PATH**.

Pour cela, faites un clic-droit sur **Poste de travail** puis **Propriétés**. Allez sur l'onglet **Avancé** puis cliquez sur **Variables d'environnement…** Editer la variable **Path** et ajouter *C:\...\j2sdk1.4.x\bin* si besoin.

3.5.1. javac

```
set CLASSPATH=C:\dossier_contenant_vos_classes
// où cd dossier_contenant_vos_classes
javac DateServer.java
javac DateServerImpl.java
javac MyServer.java
javac MyClient.java
```

3.5.2. rmic

```
set CLASSPATH=C:\dossier_contenant_vos_classes
// où cd dossier_contenant_vos_classes
rmic DateServerImpl
```

Vous pouvez remarquer qu'il ne faut pas mettre l'extension **.class**

De plus, vous devez spécifier le nom complet des classes (avec les noms de packages).

Deux nouvelles classes viennent d'être créées : **DateServerImpl_Stub** et **DateServerImpl_Skel**. Ce sont les stubs et skeletons de notre implémentation.

3.6.Lancement

```
set CLASSPATH= //pour annuler la valeur du CLASSPATH
start rmiregistry
start java MyServer
java MyClient localhost
```

Il est nécessaire d'annuler la valeur du CLASSPATH lors de chargement dynamique. Ce point sera abordé dans un prochain chapitre.

3.7.Répartitions des classes

Nous allons maintenant voir comment répartir les classes entre le serveur et le client.

Sur le poste client disposez les classes :

- MyClient.class
- DateServer.class
- DateServerImpl_Stub.class

Sur le poste serveur :

- MyServer.class
- DateServer.class
- DateServerImpl.class
- DateServerImpl_Stub.class

4. Le package java.rmi

Nous allons maintenant voir plus en détail les différents outils fournis par le package **java.rmi**.

4.1. Interface Remote

L'interface **Remote** sert à identifier les interfaces dont les méthodes pourront être invoquées depuis une JVM distante. Chaque objet distant doit implémenter cette interface directement ou indirectement. Seules les méthodes spécifiées dans une interface distante sont disponibles à distances.

Les classes d'implémentation peuvent implémenter plusieurs interfaces distantes et peuvent hériter d'autres classes d'implémentation distantes. RMI fournit quelques classes dont les objets distants peuvent hériter et qui facilitent la création d'objets distants (ex : la classe **UnicastRemoteObject**).

4.2. Classe UnicastRemoteObject

La classe **UnicastRemoteObject** définit un objet non répliqué dont les références sont valides seulement si le processus serveur est actif. Elle fournit un support pour les références à des objets actifs utilisant des flux TCP/IP.

Les objets qui exigent un comportement distant doivent étendre de **RemoteObject**, typiquement via **UnicastRemoteObject**. Dans le cas contraire, ils doivent alors assumer eux-mêmes la responsabilité des méthodes ***hashCode()***, ***equals()*** et ***toString()***. Il est donc plus simple d'utiliser la classe **UnicastRemoteObject**, car vous n'avez pas à définir vous même ces méthodes.

Le constructeur par défaut utilise le port TCP 1099. Il est toutefois possible d'utiliser un autre constructeur qui prend en paramètre un **int** correspondant au numéro de port à utiliser pour le transfert des paquets TCP.

4.3. Classe Naming

La classe **Naming** fournit des méthodes pour enregistrer et obtenir des références à des objets distants dans un registre distant (comme **rmiregistry**). Chaque méthode de la classe **Naming** prend comme argument une **String** de type URL comprenant les paramètres suivants :

- *hôte* correspond au nom (ou adresse IP) de la machine où est situé le registre
- *port* est le numéro de port sur lequel le registre écoute
- *nom* correspond au nom sous lequel l'objet distant est inscrit dans le registre

Si *hôte* est omis, il s'agira de la machine locale (localhost). Il en va de même pour le *port*, s'il est omis, ce sera le port 1099 qui sera utilisé par défaut.

Il existe plusieurs façons d'écrire l'URL :

- rmi://host:port/nom
- //host:port/nom
- //host/nom (forme la plus courante)
- //host (spécifique à la méthode ***list()***)
- nom (dans ce cas on sait que le registre est exécuté en local sur le port 1099)

Nous allons voir plus en détail les méthodes fournies par la classe **Naming**. Toutes ces méthodes sont statiques. Elles peuvent lever des exceptions de type **RemoteException** s'il y a un problème au niveau de la communication distante. De plus, elles peuvent aussi lever des exceptions de type **MalformedURLException** dans le cas où l'URL n'aurait pas la bonne syntaxe.

4.3.1. bind()

```
public static void bind(String name, Remote obj) throws AlreadyBoundException,
MalformedURLException, RemoteException
```

Cette méthode permet d'enregistrer un objet dans le registre en lui associant un nom. Elle prend en argument une **String** désignant l'URL de l'objet et un objet de type **Remote**. S'il existe déjà un lien pour cette URL dans le registre, une exception de type **AlreadyBoundException** sera levée.

4.3.2. rebind()

```
public static void rebind(String name, Remote obj) throws RemoteException,
MalformedURLException
```

Son utilisation est identique à la méthode précédente mais elle permet d'associer un nouvel objet distant au nom spécifié en paramètre. Si ce nom était déjà associé à un objet, il sera remplacé. On préférera utiliser cette méthode plutôt que la méthode ***bind()*** pour éviter les exceptions de type **AlreadyBoundException**.

4.3.3. unbind()

```
public static void unbind(String name) throws RemoteException, NotBoundException,
MalformedURLException
```

Détruit le lien entre le nom spécifié en paramètre et l'objet distant dans le registre RMI. Si le lien est inexistant, une exception de type **NotBoundException** sera levée.

4.3.4. list()

```
public static String[] list(String name) throws RemoteException,
MalformedURLException
```

Cette méthode retourne un tableau de **String** correspondant à la liste des noms de tous les liens vers des objets distants dans le registre dont l'URL est spécifiée en paramètre. L'URL spécifié est de la forme *//hôte:port*.

4.3.5. lookup()

```
public static Remote lookup(String name) throws NotBoundException,
MalformedURLException, RemoteException
```

La méthode ***lookup()*** est utilisée par les clients pour obtenir une référence à un objet distant. L'objet retourné est de type **Remote**. Cette méthode prend comme paramètre une **String** correspondant à l'URL de l'objet distant. Si le lien dans le registre n'existe pas, une exception de type **NotBoundException** sera levée.

4.4.Exceptions RemoteException

RemoteException est la super-classe commune à de nombreuses exceptions, liées à la communication, qui peuvent survenir durant l'appel d'une méthode distante. Chaque méthode d'une interface distante doit pouvoir lever des exceptions de type **RemoteException**. Il faut donc la déclarer dans la clause **throws** de la déclaration de votre méthode.

4.5.Utilisation de *rmic*

rmic est le compilateur RMI, fourni en standard avec le JDK. Il s'utilise de manière similaire à **javac** à la différence près qu'il ne faut pas spécifier l'extension des fichiers. C'est lui qui permet de générer les *stubs* et les *skeletons* des classes d'implémentations de service distant. Pour l'utiliser vous devez vous positionner dans le dossier contenant vos fichiers. Son utilisation est la suivante :

```
rmic <option> <class names>
```

où *class names* correspond aux noms complets des classes (sans l'extension). Si vos classes sont déclarées dans un package, il faut le spécifier :

```
rmic <option> nom_package.nom_classe
```

Voici les options disponibles :

- *-keep* Ne supprime pas les fichiers sources intermédiaires générés
- *-keepgenerated* (équivalent de "-keep")
- *-g* Génère des informations de debugage
- *-depend* Recompile récursivement les fichiers obsolètes
- *-nowarn* Ne génère pas d'avertissements
- *-verbose* Affiche les messages sur ce que fait le compilateur
- *-classpath <path>* Spécifie où trouver les fichiers classes et source d'entrée
- *-d <directory>* Spécifie où placer les fichiers classes générés
- *-J<runtime flag>* Passe des arguments à l'interpréteur java

5. Chargement dynamique de classe

Le framework RMI inclut aussi la distribution automatique du byte-code des *stubs*. Dans ce cas, l'application (au niveau client ou au niveau serveur) ne nécessite que ses propres classes et chaque interfaces distantes qu'il utilise. Les autres classes nécessaires faisant partie d'un appel à une méthode distante seront automatiquement téléchargées à partir d'un serveur web (FTP ou HTTP). Cette opération s'effectue à travers de la classe **RMIClassLoader**. Si un client ou un serveur exécute un système RMI et remarque qu'il doit charger une classe située à distance, il fera automatiquement appel à **RMIClassLoader** pour effectuer ce travail. Le chargement des classes par RMI est contrôlé par le paramètre **java.rmi.server.codebase** de l'interpréteur Java.

Voici la syntaxe pour définir les paramètres à l'exécution de l'interpréteur :

```
java [ -D<PropertyName>=<PropertyValue> ] <ClassFile>
```

La propriété **java.rmi.server.codebase** sert à indiquer l'URL où l'on peut trouver les fichiers à télécharger. Si un programme s'exécutant sur une JVM envoie une référence d'objet à une autre JVM (comme valeur de retour d'une méthode), cette autre JVM aura besoin de charger le fichier classe de cet objet. Lorsque RMI envoie l'objet par sérialisation, il ajoute l'URL à côté de l'objet. RMI n'envoie donc pas les fichiers classes avec les objets sérialisés.

Si la JVM distante a besoin de charger un fichier .class pour un objet, il cherche l'URL et contacte le serveur qu'elle désigne pour le fichier. Quand la propriété **java.rmi.server.useCodebaseOnly** est active, la JVM charge les classes, soit à un endroit spécifié dans le CLASSPATH, soit à l'URL spécifié par cette propriété.
En utilisant plusieurs combinaisons de propriétés système disponibles, vous pouvez créer différentes configurations système RMI.

Dans l'exemple du serveur de date, on peut imaginer que vous ayez placé le fichier **DateServerImpl_Stub.class** correspondant au stub sur un serveur web. Si vos programmes (client ou serveur) ne disposent pas de ces classes dans leur **CLASSPATH**, ils iront les chercher à l'URL que vous leur spécifier. Pour cela, il faut taper une des commandes suivantes :

```
java -Djava.rmi.server.codebase=http://172.16.X.X/path_des_classes/ MyServer
```

Si les *stubs* sont sur un serveur http.

```
java -Djava.rmi.server.codebase=ftp://172.16.X.X/path/ MyServer
```

Si les *stubs* sont sur un serveur ftp

```
java -Djava.rmi.server.codebase=http://172.16.X.X:2001/ MyServer
```

Si le serveur Web écoute sur le port TCP 2001

Dans ces exemples, on exécute le serveur mais la syntaxe est identique pour exécuter le client.

Le dernier slash (/) de l'Url est obligatoire.

L'intérêt d'utiliser un serveur web est de pouvoir modifier vos objets distants sans changer quoique ce soit sur les clients. Généralement le service web est lancé sur la même machine que le serveur RMI.

6. Stratégie de sécurité

En Java, il est possible de mettre en place une sécurité système propre à vos applications. Nous ne traiterons pas de sujet en détail car il sera traité dans un autre cours. Nous n'étudierons donc ici que la sécurité à mettre en place pour le fonctionnement de RMI.

Sur certains systèmes le chargement dynamique de classe peut être refusé pour des raisons de sécurité. Dans ce cas, une exception du type **AccessControlException** sera levée lors de l'utilisation des méthodes de la classe **Naming**.

Pour pallier ce problème, il est possible de mettre en place une stratégie de sécurité autorisant ce phénomène. Cela est possible grâce à la mise en place du gestionnaire de sécurité spécifique à RMI via l'objet **RMISecurityManager**. Il vérifie la définition des classes et autorise seulement le passage des arguments et des valeurs de retour des méthodes distantes.

Voici comment utiliser ce gestionnaire de sécurité :

```
if (System.getSecurityManager() == null)
      System.setSecurityManager(new RMISecurityManager());
```

On vérifie que le système n'a pas de gestionnaire de sécurité (pour ne pas écraser celui existant), puis on lui dit d'utiliser un objet de **RMISecurityManager**.

Ces lignes de code sont à utiliser avant de faire appel au registre par l'utilisation des méthodes de la classe **Naming**.

Vous pouvez utiliser ce gestionnaire de sécurité aussi bien du côté client que du côté serveur. Tout dépend de la nécessité de charger des classes dynamiquement.

Il vous reste maintenant à définir les différentes permissions. La solution consiste à placer vos différentes permissions dans un fichier texte spécifique. Ce fichier sera lu lors de la création du **RMISecurityManager**. Par convention, il porte l'extension **.policy**.

Voici un exemple de contenu pour ce fichier :

```
grant {
      permission java.security.Allpermission
} ;
```

Dans cet exemple, tout est permis !

C'est la permission la plus simple à configurer mais aussi la plus dangereuse.

```
grant {
      permission java.net.SocketPermission "172.16.23.32:1099", "connect";
}
```

On accepte les connexions sur le port 1099 de la machine dont l'adresse IP est 172.16.23.32

Pour lire ce fichier, vous devez dire à l'interpréteur où il peut le trouver. Cela se fait en utilisant le paramètre **java.security.policy**. Ce paramètre sera combiné avec **java.rmi.server.codebase**.

```
C:\...\> java -Djava.security.policy=path\fichier.policy
-Djava.rmi.server.codebase=http://serveur:port/path/ ClassAExecuter
```

Votre programme sait maintenant où trouver le *bytecode* des *stubs* et s'il a le droit de le récupérer.

7. Distributed Garbage Collector

RMI fournit un ramasse miette distribué. Son intérêt est le même que le *Garbage Collector* local.

Il sert à supprimer les objets distants qui ne sont plus référencés par des clients. Il est activé lorsque plus aucun client ne possède de souche (*stub*) de l'objet distant.

Il est basé sur un système de compteur de référence des souches et sur l'interaction des *Garbage Collector* côté client et côté serveur.

Lorsqu'un ramasse miette supprime une souche cela entraîne la décrémentation du compteur de référence du serveur. Si le compteur est à 0, le *Garbage Collector* du serveur supprime l'objet.

Il est possible de signaler aux objets qu'ils ne sont plus référencés. Pour cela ils doivent implémenter l'interface **Unreferenced**. Dans ce cas, ils doivent implémenter la méthode ***unreferenced()***. Celle-ci sera appelée avant la destruction de l'objet. C'est le même principe que pour l'appel de la méthode ***finalize()*** pour un objet standard.

```
import java.rmi.*;
import java rmi.server.*;

public class ObjetDistantImpl extends UnicastRemoteObjet implements ObjetDistant,
Unreferenced {

	public ObjetDistantImpl() throws RemoteException {
	}

	public void unreferenced() {
		System.out.println("L'objet n'est plus référencé.");
	}

	public void methodeDistante() {
	}
}
```